

# Second-Order Propositional Satisfiability

Cristina David, Daniel Kroening, and Matt Lewis

University of Oxford

**Abstract.** Fundamentally, every static program analyser searches for a proof through a combination of heuristics providing candidate solutions and a candidate validation technique. Essentially, the heuristic reduces a second-order problem to a first-order/propositional one, while the validation is often just a call to a SAT/SMT solver. This results in a monolithic design of such analyses that conflates the formulation of the problem with the solving process. Consequently, any change to the latter causes changes to the whole analysis. This design is dictated by the state of the art in solver technology. While SAT/SMT solvers have experienced tremendous progress, there are barely any second-order solvers. This paper takes a step towards addressing this situation by proposing a decidable fragment of second-order logic that is still expressive enough to capture numerous program analysis problems (e.g. safety proving, bug finding, termination and non-termination proving, superoptimisation). We refer to the satisfiability problem for this fragment as Second-Order SAT and show it is NEXPTIME-complete. Finally, we build a decision procedure for Second-Order SAT based on program synthesis and present experimental evidence that our approach is tractable for program analysis problems.

**Keywords:** Second-order logic, decision procedure, program synthesis, bitvectors.

## 1 Introduction

Fundamentally, every static program analysis is searching for a *program proof*. For safety analysers this proof takes the form of a program invariant [1], for bug finders it's a counter-model [2], for termination analysis it can be a ranking function [3], whereas for non-termination it's a recurrence set [4]. Finding each of these proofs was subject to extensive research resulting in a multitude of techniques.

The process of searching for a proof can be roughly seen as a refinement loop with two phases. One phase is heuristic in nature, e.g. adjusting the unwinding depth (for bounded model checking [2]), refining the set of predicates (for predicate abstraction and interpolation [5,6]), selecting a template (for template-based analyses [7]), applying a widening operator (for abstract interpretation based techniques [1]), whereas the other phase usually involves a call to a decision procedure. From the perspective of the proof, the heuristic constrains the universe of potential proofs to just one candidate (by fixing the unwinding bound, the

template, the set of predicates, etc), which is then validated by the other phase. The unknowns in the first phase are proofs (second-order entities), whereas the unknowns in the second phase are program variables. Essentially, the first phase reduces a second-order problem to a first-order/propositional one.

Such a design makes it difficult to separate the problem’s formulation (the second-order problem) from parts of the solving process, resulting in analyses that are cluttered and fragile. Any change to the search process causes changes to the whole analysis. Ideally, we would like a modular design, where the search for a solution is encapsulated and, thus, completely separated from the formulation of the problem.

The existing design is dictated by the state of the art in solver technology. While the SAT/SMT technologies nowadays allow solving industrial sized instances, there is hardly any progress made for second-order solvers.

*Aim of current paper.* In this paper, we take a step towards addressing this situation by presenting a fragment of second-order logic that is decidable, while still expressive enough to capture static analysis problems, such as safety proving, bug finding, termination and non-termination proving, superoptimisation, etc. Decidability is regained by allowing only existential second-order quantification and interpreting ground terms over a finite universe. We call the resulting problem Second-Order SAT and show it is NEXPTIME-complete.

By using finite state program synthesis, we build a solver for Second-Order SAT. A very important property of our solver is that its runtime is heavily influenced by the *length of the shortest proof*, i.e. the Kolmogorov complexity of the problem as we will discuss in section 6. If a short proof exists, then the solver will find it quickly. This is particularly useful for program analysis problems, where, if a proof exists, then most of the time many proofs exist and some are short ([8] relies on a similar remark about loop invariants). Our experiments show that for many problems the shortest solution is often very short, which means that we avoid the NEXPTIME bound, so our approach is tractable far more often than one might reasonably expect.

In the same way that SAT solvers enabled a leap forwards by presenting a unified interface to an NP-complete problem, Second-Order SAT presents a unified interface for NEXPTIME problems. For program analysers, this means that the business of finding a solution (running a refinement loop) can be offloaded to a black box solver. We advocate for using second-order logic for formulating static analysis problems, and hope that this work will encourage researchers in this direction. Analyses formulated this way will automatically benefit from future advances in second-order solving.

*Related logics.* Other second-order solvers are introduced in [9,10]. As opposed to Second-Order SAT, these are specialised for Horn clauses and the logic they handle is undecidable. Wintersteiger et al. present in [11] a decision procedure for a logic related to Second-Order SAT, the Quantified bit-vector logic, which is a many sorted first-order logic formula where the sort of every variable is a bit-vector sort.

*Technical Contributions:*

- We define the Second-Order SAT problem and show that it is NEXPTIME-complete.
- We show that Second-Order SAT is expressive enough to encode several program analysis problems.
- We build a decision procedure for Second-Order SAT via a reduction to finite state program synthesis. The decision procedure uses a novel combination of symbolic model checking, explicit state model checking and stochastic search, and its runtime is dependent on the length of the shortest proof.
- We evaluated our solver on several static analysis problems.

## 2 Preliminaries

In this section we will recall some well known decision problems along with their associated complexity classes.

**Definition 1 (Propositional SAT).**

$$\exists x_1 \dots x_n. \sigma$$

Where the  $x_i$  range over Boolean values and  $\sigma$  is a quantifier-free propositional formula whose free variables are the  $x_i$ .

Checking the truth of an instance of Definition 1 is NP-complete.

**Definition 2 (First-Order Propositional SAT or QBF).**

$$Q_1 x_1. Q_2 x_2 \dots Q_n x_n. \sigma$$

Where the  $Q_i$  are either  $\exists$  or  $\forall$ . The  $x_i$  and  $\sigma$  are as in Definition 1.

Checking the truth of an instance of Definition 2 is PSPACE-complete.

## 3 Second-Order SAT

In this section, we introduce Second-Order SAT, an extension of propositional SAT. Subsequently, we prove that Second-Order SAT is NEXPTIME-complete.

**Definition 3 (Second-Order SAT).**

$$\exists S_1 \dots S_m. Q_1 x_1 \dots Q_n x_n. \sigma$$

Where the  $S_i$  range over predicates. Each  $S_i$  has an associated arity  $\text{ar}(S_i)$  and  $S_i \subseteq \mathbb{B}^{\text{ar}(S_i)}$ . The remainder of the formula is an instance of Definition 2, except that the quantifier-free part ( $\sigma$ ) may refer to both the first-order variables  $x_i$  and the second-order variables  $S_i$ .

*Example 1.* The following is a Second-Order SAT formula:

$$\exists S. \forall x_1, x_2. S(x_1, x_2) \rightarrow S(x_2, x_1)$$

This formula is satisfiable and is satisfied by any symmetric relation.

**Theorem 1 (Fagin’s Theorem [12]).** *The class of structures  $A$  recognisable in time  $|A|^k$ , for some  $k$ , by a nondeterministic Turing machine is exactly the class of structures definable by existential second-order sentences.*

**Theorem 2 (Second-Order SAT is NEXPTIME-complete).** *For an instance of Definition 3 with  $n$  first-order variables, checking the truth of the formula is NEXPTIME-complete.*

*Proof.* We will apply Theorem 1. To do so we must establish the size of the universe implied by Theorem 1. Since Definition 3 uses  $n$  Boolean variables, the universe is the set of interpretations of  $n$  Boolean variables. This set has size  $2^n$ , and so by Theorem 1, Definition 3 defines exactly the class sets recognisable in  $(2^n)^k$  time by a nondeterministic Turing machine. This is the class NEXPTIME, and so checking validity of an arbitrary instance of Definition 3 is NEXPTIME-complete.

For an alternative proof, consider a Turing machine  $M$ . For a particular run of  $M$  we can construct a relation  $f(k, q, h, j, t)$  defined such that after  $k$  steps  $M$  is in state  $q$ , with its head at position  $h$  and tape cell  $j$  containing the symbol  $t$ . If  $M$  halts within  $2^n$  steps on an input of length  $n$ , the values of all the variables in this relation are bounded by  $2^n$ , which means they can be written down using  $n$  bits. The details of creating a first-order formula constraining  $f$  to reflect the behaviour of  $M$  are left to the reader.

## 4 Decision Procedure for Second-Order SAT

A solution to an instance of Definition 3 is an assignment mapping each of the second-order variables to some function of the appropriate type and arity. When deciding whether a particular Second-Order SAT instance is satisfiable, we should think about how solutions are encoded and in particular how a function is to be encoded. The functions all have a finite domain and co-domain, so their canonical representation would be a finite set of ordered pairs. Such a set is exponentially large in the size of the domain, so we would prefer to work with a more compact representation if possible.

We will generate *finite state programs* that compute the functions and represent these programs as finite lists of instructions in SSA form. This representation has the following properties, proofs for which can be found in Appendix A.

**Theorem 3.** *Every total, finite function is computed by at least one program in this language.*

**Theorem 4.** *Furthermore, this representation is optimally concise – there is no encoding that gives a shorter representation to every function.*

*Finite State Program Synthesis* To formally define the finite state synthesis problem, we need to fix some notation. We will say that a program  $P$  is a finite list of instructions in SSA form, where no instruction can cause a back jump, i.e. our programs are loop free and non-recursive. Inputs  $x$  to the program are drawn from some finite domain  $\mathcal{D}$ . The synthesis problem is given to us in the form of a specification  $\sigma$  which is a function taking a program  $P$  and input  $x$  as parameters and returning a boolean telling us whether  $P$  did “the right thing” on input  $x$ . Basically, the finite state synthesis problem checks the truth of Definition 4.

**Definition 4 (Finite Synthesis Formula).**

$$\exists P. \forall x \in \mathcal{D}. \sigma(P, x)$$

To express the specification  $\sigma$ , we introduce a function  $\mathbf{exec}(P, x)$  that returns the result of running program  $P$  with input  $x$ . Since  $P$  cannot contain loops or recursion,  $\mathbf{exec}$  is a total function.

*Example 2.* The following finite state synthesis problem is satisfiable:

$$\exists P. \forall x \in \mathbb{N}_8. \mathbf{exec}(P, x) \geq x$$

One such program  $P$  satisfying the specification is `return 8`, which just returns 8 for any input.

We now present our main theorem, which says that Second-Order SAT can be reduced to finite state program synthesis. The proof of this theorem can be found in Appendix A.

**Theorem 5 (Second-Order SAT is Polynomial Time Reducible to Finite Synthesis).** *Every instance of Definition 3 is polynomial time reducible to an instance of Definition 4.*

**Corollary 1.** *Finite-state program synthesis is NEXPTIME-complete.*

We are now in a position to sketch the design of a decision procedure for Second-Order SAT: we will convert the Second-Order SAT problem to an equisatisfiable finite synthesis problem which we will then solve with a finite state program synthesiser. This design will be elaborated in Section 5.

## 5 Solving Second-Order SAT via Finite-State Program Synthesis

In this section we will present a sound and complete algorithm for finite-state synthesis that we use to decide Second-Order SAT. We begin by describing a general purpose synthesis procedure (Section 5.1), then detail how this general purpose procedure is instantiated for synthesising finite-state programs. We then describe the algorithm we use to search the space of possible programs (Sections 5.4, 5.5 and 5.6).

### 5.1 General Purpose Synthesis Algorithm

We use Counterexample Guided Inductive Synthesis (CEGIS) [13,14,15] to find a program satisfying our specification. The core of the CEGIS algorithm is the refinement loop detailed in Algorithm 1.

---

**Algorithm 1** Abstract refinement algorithm

---

<pre> 1: <b>function</b> SYNTH(inputs) 2:   <math>(i_1, \dots, i_N) \leftarrow \text{inputs}</math> 3:   <math>\text{query} \leftarrow \exists P. \sigma(i_1, P) \wedge \dots \wedge \sigma(i_N, P)</math> 4:   <math>\text{result} \leftarrow \text{decide}(\text{query})</math> 5:   <b>if</b> <math>\text{result.satisfiable}</math> <b>then</b> 6:     <b>return</b> <math>\text{result.model}</math> 7:   <b>else</b> 8:     <b>return</b> UNSAT </pre>	<pre> 16: <b>function</b> REFINEMENT LOOP 17:   <math>\text{inputs} \leftarrow \emptyset</math> 18:   <b>loop</b> 19:     <math>\text{candidate} \leftarrow \text{SYNTH}(\text{inputs})</math> 20:     <b>if</b> <math>\text{candidate} = \text{UNSAT}</math> <b>then</b> 21:       <b>return</b> UNSAT 22:     <math>\text{res} \leftarrow \text{VERIF}(\text{candidate})</math> 23:     <b>if</b> <math>\text{res} = \text{valid}</math> <b>then</b> 24:       <b>return</b> <math>\text{candidate}</math> 25:     <b>else</b> 26:       <math>\text{inputs} \leftarrow \text{inputs} \cup \text{res}</math> </pre>
<pre> 9: <b>function</b> VERIF(P) 10:  <math>\text{query} \leftarrow \exists x. \neg \sigma(x, P)</math> 11:  <math>\text{result} \leftarrow \text{decide}(\text{query})</math> 12:  <b>if</b> <math>\text{result.satisfiable}</math> <b>then</b> 13:    <b>return</b> <math>\text{result.model}</math> 14:  <b>else</b> 15:    <b>return</b> valid </pre>	

---

The algorithm is divided into two procedures: SYNTH and VERIF, which interact via a finite set of test vectors INPUTS. The SYNTH procedure tries to find an existential witness  $P$  that satisfies the partial specification:

$$\exists P. \forall x \in \text{INPUTS}. \sigma(x, P)$$

If SYNTH succeeds in finding a witness  $P$ , this witness is a candidate solution to the full synthesis formula. We pass this candidate solution to VERIF which determines whether it does satisfy the specification on all inputs by checking satisfiability of the verification formula:

$$\exists x. \neg \sigma(x, P)$$

If this formula is unsatisfiable, the candidate solution is in fact a solution to the synthesis formula and so the algorithm terminates. Otherwise, the witness  $x$  is an input on which the candidate solution fails to meet the specification. This witness  $x$  is added to the INPUTS set and the loop iterates again. It is worth noting that each iteration of the loop adds a new input to the set of inputs being used for synthesis. If the full set of inputs  $X$  is finite, this means that the refinement loop can only iterate a finite number of times.

## 5.2 Finite-State Synthesis

We will now show how the generic construction of Section 5.1 can be instantiated to produce a useful finite-state program synthesiser. A natural choice for such a synthesiser would be to work in the logic of quantifier-free propositional formulae and to use a propositional SAT or SMT- $\mathcal{BV}$  solver as the decision procedure. However we propose a slightly different tack, which is to use a decidable fragment of  $C$  as a “high level” logic. We call this fragment  $C^-$ .

The characteristic property of a  $C^-$  program is that safety can be decided for it using a single query to a Bounded Model Checker. A  $C^-$  program is just a  $C$  program with the following syntactic restrictions:

- all loops in the program must have a constant bound;
- all recursion in the program must be limited to a constant depth;
- all arrays must be statically allocated (i.e. not using `malloc`), and be of constant size.

$C^-$  programs may use nondeterministic values, assumptions and arbitrary-width types.

Since each loop is bounded by a constant, and each recursive function call is limited to a constant depth, a  $C^-$  program necessarily terminates and in fact does so in  $O(1)$  time. If we call the largest loop bound  $k$ , then a Bounded Model Checker with an unrolling bound of  $k$  will be a complete decision procedure for the safety of the program. For a  $C^-$  program of size  $l$  and with largest loop bound  $k$ , a Bounded Model Checker will create a SAT problem of size  $O(lk)$ . Conversely, a SAT problem of size  $s$  can be converted trivially into a loop-free  $C^-$  program of size  $O(s)$ . The safety problem for  $C^-$  is therefore NP-complete, which means it can be decided fairly efficiently for many practical instances.

## 5.3 Encoding the Problem in $C^-$

To instantiate the abstract synthesis algorithm in  $C^-$ , we must express  $X, Y, \sigma$  in  $C^-$ , then express the validity of the synthesis formula as a safety property of the resulting  $C^-$  program.

Our encoding for these pieces is the following:

- $X$  is the set of  $N$ -tuples of 32-bit bitvectors.
- $Y$  is the set of  $M$ -tuples of 32-bit bitvectors.
- $\sigma$  is a pure function with type  $X \times Y \rightarrow \text{Bool}$ .

$P$  is written in a simple RISC-like language  $\mathcal{L}$ , whose syntax is given in Fig. 1. Programs in  $\mathcal{L}$  have type  $X \rightarrow Y$ . We supply an interpreter for  $\mathcal{L}$  which is written in  $C^-$ . The type of this interpreter is  $(X \rightarrow Y) \times X \rightarrow Y$ . The specification function  $\sigma$  will include calls to this interpreter, by which means it will examine the behaviour of a candidate  $\mathcal{L}$  program.

With these pieces in place, we construct a  $C^-$  program `SYNTH.C` which takes as parameters a candidate program  $P$  and test inputs  $X$ . The program contains an assertion which fails iff  $P$  meets the specification for each of the inputs in  $X$ . Finding a new candidate program is then equivalent to checking the safety of `SYNTH.C` for which we use the strategies described in the next section.

Integer arithmetic instructions:

add a b	sub a b	mul a b	div a b
neg a	mod a b	min a b	max a b

Bitwise logical and shift instructions:

and a b	or a b	xor a b
lshr a b	ashr a b	not a

Unsigned and signed comparison instructions:

le a b	lt a b	sle a b
slt a b	eq a b	neq a b

Miscellaneous logical instructions:

implies a b	ite a b c
-------------	-----------

Floating-point arithmetic:

fadd a b	fsub a b	fmul a b	fdiv a b
----------	----------	----------	----------

Fig. 1: The language  $\mathcal{L}$

#### 5.4 Candidate Generation Strategies

The remit of the SYNTH portion of the CEGIS loop is to generate candidate programs. There are many possible strategies for finding these candidates; we employ the following strategies in parallel:

*Explicit Proof Search.* The simplest strategy for finding candidates is to just exhaustively enumerate them all, starting with the shortest and progressively increasing the number of instructions. Since the set of  $\mathcal{L}$ -programs is recursively enumerable, this procedure is complete.

*Symbolic Bounded Model Checking.* Another complete method for generating candidates is to simply use BMC on the SYNTH.C program. As with explicit search, we must progressively increase the length of the  $\mathcal{L}$ -program we search for in order to get a complete search procedure.

*Genetic Programming and Incremental Evolution.* Our final strategy is genetic programming (GP) [16,17]. GP provides an adaptive way of searching through the space of  $\mathcal{L}$ -programs for an individual that is “fit” in some sense. We measure the fitness of an individual by counting the number of tests in INPUTS for which it satisfies the specification.

To bootstrap GP in the first iteration of the CEGIS loop, we generate a population of random  $\mathcal{L}$ -programs. We then iteratively evolve this population by applying the genetic operators CROSSOVER and MUTATE. CROSSOVER combines selected existing programs into new programs, whereas MUTATE randomly changes parts of a single program. Fitter programs are more likely to be selected.

Rather than generating a random population at the beginning of each subsequent iteration of the CEGIS loop, we start with the population we had at



the end of the previous iteration. The intuition here is that this population contained many individuals that performed well on the  $k$  inputs we had before, so they will probably continue to perform well on the  $k + 1$  inputs we have now. In the parlance of evolutionary programming, this is known as incremental evolution [18].

### 5.5 Parameterising the Program Space

In order to search the space of candidate programs, we parametrise the language  $\mathcal{L}$ , inducing a lattice of progressively more expressive languages. We start by attempting to synthesise a program at the lowest point on this lattice and increase the parameters of  $\mathcal{L}$  until we reach a point at which the synthesis succeeds.

As well as giving us an automatic search procedure, this parametrisation greatly increases the efficiency of our system since languages low down the lattice are very easy to decide safety for. If a program can be synthesised in a low-complexity language, the whole procedure finishes much faster than if synthesis had been attempted in a high-complexity language.

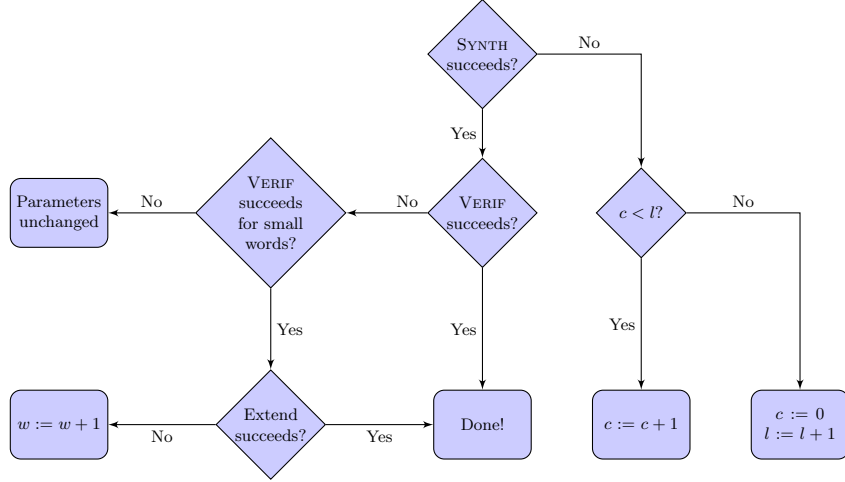
*Program Length:  $l$*  The first parameter we introduce is program length, denoted by  $l$ . At each iteration we synthesise programs of length exactly  $l$ . We start with  $l = 1$  and increment  $l$  whenever we determine that no program of length  $l$  can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct.

*Word Width:  $w$*  An  $\mathcal{L}$ -program runs on a virtual machine (the  $\mathcal{L}$ -machine) that has its own set of parameters. The only relevant parameter is the *word width* of the  $\mathcal{L}$ -machine, that is, the number of bits in each internal register and immediate constant. This parameter is denoted by  $w$ . The size of the final SAT problem generated by CBMC scales polynomially with  $w$ , since each intermediate C variable corresponds to  $w$  propositional variables.

*Number of Constants:  $c$*  Instructions in  $\mathcal{L}$  take either one or two operands. Since any instruction whose operands are all constants can always be eliminated (since its result is a constant), we know that a loop-free program of minimal length will not contain any instructions with two constant operands. Therefore the number of constants that can appear in a minimal program of length  $l$  is at most  $l$ . By minimising the number of constants appearing in a program, we are able to use a particularly efficient program encoding that speeds up the synthesis procedure substantially. The number of constants used in a program is the parameter  $c$ .

### 5.6 Searching the Program Space

The key to our automation approach is to come up with a sensible way in which to adjust the  $\mathcal{L}$ -parameters in order to cover all possible programs. After each

Fig. 2: Decision tree for increasing parameters of  $\mathcal{L}$ .

round of SYNTH, we may need to adjust the parameters. The logic for these adjustments is shown as a tree in Fig. 2.

Whenever SYNTH fails, we consider which parameter might have caused the failure. There are two possibilities: either the program length  $l$  was too small, or the number of allowed constants  $c$  was. If  $c < l$ , we just increment  $c$  and try another round of synthesis, but allowing ourselves an extra program constant. If  $c = l$ , there is no point in increasing  $c$  any further. This is because no minimal  $\mathcal{L}$ -program has  $c > l$ , for if it did there would have to be at least one instruction with two constant operands. This instruction could be removed (at the expense of adding its result as a constant), contradicting the assumed minimality of the program. So if  $c = l$ , we set  $c$  to 0 and increment  $l$ , before attempting synthesis again.

If SYNTH succeeds but VERIF fails, we have a candidate program that is correct for some inputs but incorrect on at least one input. However, it may be the case that the candidate program is correct for *all* inputs when run on an  $\mathcal{L}$ -machine with a small word size. For example, we may have synthesised a program which is correct for all 8-bit inputs, but incorrect for some 32-bit input. If this is the case (which we can determine by running the candidate program through VERIF using the smaller word size), we may be able to produce a correct program for the full  $\mathcal{L}$ -machine by using some constant extension rules. If constant generalization is able to find a correct program, we are done. Otherwise, we need to increase the word width of the  $\mathcal{L}$ -machine we are currently synthesising for.

### 5.7 Stopping Condition for Unsatisfiable Specifications

If a specification is unsatisfiable, we would still like our algorithm to terminate with an “unsatisfiable” verdict. To do this, we can observe that any total function taking  $n$  bits of input is computed by some program of at most  $2^n$  instructions (a consequence of Theorems 3 and 4). Therefore every satisfiable specification has a solution with at most  $2^n$  instructions. This means that if we ever need to increase the length of the candidate program we search for beyond  $2^n$ , we can terminate, safe in the knowledge that the specification is unsatisfiable.

## 6 Soundness and Completeness

We will now state soundness and completeness results for the Second-Order SAT solver. Proofs for each of these theorems can be found in Appendix A.

**Theorem 6.** *Algorithm 1 is sound – if it terminates with witness  $P$ , then  $P \models \sigma$ .*

**Theorem 7.** *Algorithm 1 is semi-complete – if a solution  $P \models \sigma$  exists then Algorithm 1 will find it.*

**Theorem 8.** *Algorithm 1 with the stopping condition described in Section 5.7 is complete when instantiated with  $C^-$  as a background theory – it will terminate for all specifications  $\sigma$ .*

Since safety of  $C^-$  programs is decidable, Algorithm 1 is sound and complete when instantiated with  $C^-$  as a background theory and using the stopping condition of Section 5.7. This construction therefore gives us a decision procedure for Second-Order SAT.

*Runtime as a Function of Solution Size.* We note that the runtime of our solver is heavily influenced by the length of the shortest program satisfying the specification, since we begin searching for short programs. We will now show that the number of iterations of the CEGIS loop is a function of the Kolmogorov complexity of the synthesised program. Let us first recall the definition of the Kolmogorov complexity of a function  $f$ :

**Definition 5 (Kolmogorov complexity).** *The Kolmogorov complexity  $K(f)$  is the length of the shortest program that computes  $f$ .*

We can extend this definition slightly to talk about the Kolmogorov complexity of a synthesis problem in terms of its specification:

**Definition 6 (Kolmogorov complexity of a synthesis problem).** *The Kolmogorov complexity of a program specification  $K(\sigma)$  is the length of the shortest program  $P$  such that  $P$  is a witness to the satisfiability of  $\sigma$ .*

Let us consider the number of iterations of the CEGIS loop  $n$  required for a specification  $\sigma$ . Since we enumerate candidate programs in order of length, we are always synthesising programs with length no greater than  $K(\sigma)$  (since when we enumerate the first correct program, we will terminate). So the space of solutions we search over is the space of functions computed by  $\mathcal{L}$ -programs of length no greater than  $K(\sigma)$ . Let's denote this set  $\mathcal{L}(K(\sigma))$ . Since there are  $O(2^{K(\sigma)})$  programs of length  $K(\sigma)$  and some functions will be computed by more than one program, we have  $|\mathcal{L}(K(\sigma))| \leq O(2^{K(\sigma)})$ .

Each iteration of the CEGIS loop distinguishes at least one incorrect function from the set of correct functions, so the loop will iterate no more than  $|\mathcal{L}(K(\sigma))|$  times. Therefore another bound on our runtime is  $N\text{TIME}(2^{K(\sigma)})$ .

## 7 Applications of Second-Order SAT

Program analysis problems can be reduced to the problem of finding solutions to a second-order constraint [19,9,20]. In this section we will show that, although decidable, Second-Order SAT is still expressive enough to capture many interesting such problems. Moreover, since Second-Order SAT is NEXPTIME-complete, any NEXPTIME problem can be solved with a Second-Order SAT solver. When we describe analyses related to loops, we will characterise the loop as having initial state  $I$ , guard  $G$ , transition relation  $B$ .

*Safety Invariants.* Given a safety assertion  $A$ , a safety invariant is a set of states  $S$  which is inductive with respect to the program's transition relation, and which excludes an error state. A predicate  $S$  is a safety invariant iff it satisfies the following criteria:

$$\exists S. \forall x, x'. I(x) \rightarrow S(x) \wedge \quad (1)$$

$$S(x) \wedge G(x) \wedge B(x, x') \rightarrow S(x') \wedge \quad (2)$$

$$S(x) \wedge \neg G(x) \rightarrow A(x) \quad (3)$$

1 says that each state reachable on entry to the loop is in the set  $S$ , and in combination with 2 shows that every state that can be reached by the loop is in  $S$ . The final criterion 3 says that if the loop exits while in an  $S$ -state, the assertion  $A$  is not violated.

*Termination and non-termination.* As shown in [20], termination of a loop can be encoded as the following Second-Order SAT formula, where  $W$  is an inductive invariant of the loop that is established by the initial states  $I$  if the loop guard  $G$  is met, and  $R$  is a ranking function as restricted by  $W$ :

$$\exists R, W. \forall x, x'. I(x) \wedge G(x) \rightarrow W(x) \wedge$$

$$G(x) \wedge W(x) \wedge B(x, x') \rightarrow W(x') \wedge R(x) > 0 \wedge R(x) > R(x')$$

Similarly, non-termination can be expressed in Second-Order SAT as follows:

$$\begin{aligned} \exists N, C, x_0. \forall x. N(x_0) \wedge N(x) \rightarrow G(x) \wedge \\ N(x) \rightarrow B(x, C(x)) \wedge N(C(x)) \end{aligned}$$

$N$  denotes a recurrence set, i.e. a nonempty set of states such that for each  $s \in N$  there exists a transition to some  $s' \in N$ , and  $C$  is a Skolem function that chooses the successor  $x'$ . More details on the formulations for termination and non-termination can be found in [20].

*Other NEXPTIME Problems.* NEXPTIME is a very large complexity class, which includes decision problems over infinite-state and finite-state systems. Some other interesting problems in NEXPTIME, and which are therefore reducible to Second-Order SAT, include:

- Satisfiability of modal- $\mu$  calculus [21].
- Reachability of pushdown systems [22].
- Finding a winning strategy for Go [23].

Problems in any of the classes included in NEXPTIME are of course reducible to Second-Order SAT. For instance, we have evaluated our solver on the Quantified Boolean Formula (QBF) problem, a well-known PSPACE-complete problem.

## 8 Experiments

We implemented our decision procedure for Second-Order SAT as the KALASHNIKOV tool. To evaluate the viability of Second-Order SAT, we used KALASHNIKOV to solve formulae generated from a variety of problems. Our benchmarks come from superoptimisation, code deobfuscation, floating point verification, ranking function and recurrent set synthesis, and QBF solving. The superoptimisation and code deobfuscation benchmarks were taken from the experiments of [24]; the termination benchmarks were taken from SVCOMP'15 [25] and they include the experiments of [20]; the QBF instances consist of some simple instances created by us and some harder instances taken from [26].

We would like to stress that these experiments serve to evaluate the potential of using Second-Order SAT as a backend for many program analysis tasks and are not intended to compare performance with specialised solvers for each of these tasks.

We ran our experiments on a 4-core, 3.30 GHz Core i5 with 8 GB of RAM. Each benchmark was run with a timeout of 180s. The results are shown in Table 1. For each category of benchmarks, we report the total number of benchmarks in that category, the number we were able to solve within the time limit, the average specification size (in lines of code), the average solution size (in instructions), the average number of iterations of the CEGIS loop, the average

time and total time taken. The deobfuscation and floating point benchmarks are considered together with the superoptimisation ones.

It should be understood that in contrast to less expressive logics that might be invoked several times in the analysis of some problem, each of these benchmarks is a “complete” problem from the given problem domain. For example, each of the benchmarks in the termination category requires KALASHNIKOV to prove that a full program terminates, i.e. it must find a ranking function and supporting invariants, then prove that these constitute a valid termination proof for the program being analysed.

*Discussion of the experimental results.* The timings show that for the instances where we can find a satisfying assignment, we tend to do so quite quickly (on the order of a few seconds). Furthermore the programs we synthesise are often short, even when the problem domain is very complex, such as for termination or QBF.

Not all of these benchmarks are satisfiable, and in particular around half of the termination benchmarks correspond to attempted proofs that non-terminating programs terminate and vice versa. This illustrates one of the current shortcomings of Second-Order SAT as a decision procedure: we can only conclude that a formula is unsatisfiable once we have examined candidate solutions up to a very high length bound. Being able to detect unsatisfiability of a Second-Order SAT formula earlier than this would be extremely valuable. We note that for some formulae we can simultaneously search for a proof of satisfiability and of unsatisfiability. For example, since QBF is closed under negation, we can take a QBF formula  $\phi$  then encode both  $\phi$  and  $\neg\phi$  as second-order SAT formulae which we then solve.

Category	#Benchmarks	#Solved	Spec. size	Solution size	Iterations	Avg. time (s)	Total time (s)
Superoptimisation	29	22	19.0	4.1	2.7	7.9	166.1
Termination	78	33	93.5	5.7	14.4	11.8	390.4
QBF (simple)	4	4	12.2	9	1.0	1.8	7.1
QBF (hard)	7	1	5889.0	11.0	2.0	1.5	1.5
Total	113	59	49116	295	536	—	565.2

Table 1: Experimental results.

To help understand the role of the different solvers involved in the synthesis process, we provide a breakdown of how often each solver “won”, i.e. was the first to return an answer. This breakdown is shown in Table 2a. We see that GP and explicit account for the great majority of the responses, with the load spread fairly evenly between them. This distribution illustrates the different strengths of each solver: GP is very good at generating candidates, explicit is very good at finding counterexamples and CBMC is very good at proving that candidates are correct. The GP and explicit numbers are similar because they are approximately “number of candidates found” and “number of candidates

refuted” respectively. The CBMC column is approximately “number of candidates proved correct”. The spread of winners here shows that each of the search strategies is contributing something to the overall search and that the strategies are able to co-operate with each other.

CBMC	Explicit	GP	Total
140	510	504	1183
12%	46%	42%	100%

(a) How often each solver “wins”.

SYNTH	VERIF	GENERALIZE	Total
389.2 s	175.8 s	25.6 s	565.2 s
69%	31%	5%	100%

(b) Where the time is spent.

Table 2: Statistics about the experimental results.

To help understand where the time is spent in our solver, Table 2b shows how much time is spent in SYNTH, VERIF and constant generalization. Note that generalization counts towards VERIF’s time. We can see that synthesising candidates takes longer than verifying them, but the ratio of around 2:1 is interesting in that neither phase completely dominates the other in terms of runtime cost. This suggests there is great potential in optimising either of these phases.

## 9 Conclusions and Future Work

We have shown that Second-Order SAT is a very expressive logic occupying a high complexity class. We have also demonstrated that it is well suited to program verification by directly encoding safety and liveness properties as Second-Order SAT formulae. Moreover, other applications, such as superoptimisation and QBF solving, map naturally onto Second-Order SAT.

We built a decision procedure for Second-Order SAT problem via a reduction to finite state program synthesis. The synthesis algorithm is novel and uses a combination of symbolic model checking, explicit state model checking and stochastic search. We have observed that for such a difficult problem, surprisingly many instances can be solved fairly rapidly. This is because the runtime of the solver is strongly influenced by the size of the shortest solution to the problem, and it seems that many real-world problems have short solutions.

*Future Work.* There is a lot of scope for continuing work on Second-Order SAT. There are many more problems that can be encoded as Second-Order SAT, and many opportunities for building more efficient solvers. One potential application is generalisation. We note that many program analysis problems are concerned with finding properties of a program that can be generalised, then finding ways of doing such a generalisation. We conjecture that *every* successful generalisation strategy would correspond to going from a long proof to a shorter one, and so consequently that there is a link between the class of tractable program analysis problems and the class of Second-Order SAT problems with short solutions.

## References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977) 238–252
2. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and verilog programs using bounded model checking. In: DAC. (2003) 368–371
3. Floyd, R.W.: Assigning meanings to programs. *Mathematical Aspects of Computer Science* (1967)
4. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.G.: Proving non-termination. In: POPL. (2008) 147–158
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: *Computer Aided Verification, CAV*. (2000) 154–169
6. McMillan, K.L.: Lazy abstraction with interpolants. In: CAV. Volume 4144 of LNCS., Springer (2006) 123–136
7. Leike, J., Heizmann, M.: Ranking templates for linear loops. In: TACAS. (2014) 172–186
8. Kong, S., Jung, Y., David, C., Wang, B., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: *Programming Languages and Systems - Asian Symposium, APLAS*. (2010) 328–343
9. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. (2012) 405–416
10. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified horn clauses. In: CAV. (2013) 869–882
11. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: *Formal Methods in Computer-Aided Design, FMCAD*. (2010) 239–246
12. Fagin, R.: Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In Karp, R., ed.: *Complexity of Computation*. Amer Mathematical Society (June 1974) 43–73
13. Solar Lezama, A.: Program Synthesis By Sketching. PhD thesis, EECS Department, University of California, Berkeley (Dec 2008)
14. Solar-Lezama, A.: Program sketching. STTT **15**(5-6) (2013) 475–495
15. Brain, M., Crick, T., Vos, M.D., Fitch, J.: TOAST: Applying answer set programming to superoptimisation. In: ICLP. (2006) 270–284
16. Langdon, W.B., Poli, R.: *Foundations of Genetic Programming*. Springer (2002)
17. Brameier, M., Banzhaf, W.: *Linear Genetic Programming*. Genetic and Evolutionary Computation. Springer (2007)
18. Gomez, F., Miikkulainen, R.: Incremental evolution of complex general behavior. *Adaptive Behavior* **5** (1997) 5–317
19. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation*, Tucson, AZ, USA, June 7-13, 2008. (2008) 281–292
20. David, C., Kroening, D., Lewis, M.: Unrestricted termination and non-termination proofs for bit-vector programs. In: ESOP. (2015)
21. Kozen, D.: Results on the propositional  $\mu$ -calculus. *Theoretical computer science* **27**(3) (1983) 333–354
22. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: *CONCUR'97: Concurrency Theory*. Springer (1997) 135–150



- 23. Robson, J.M.: The complexity of Go. In: IFIP Congress. (1983) 413–417
- 24. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI. (2011) 62–73
- 25. : <http://sv-comp.sosy-lab.org/2015/>.
- 26. Giunchiglia, E., Narizzano, M., Pulina, L., Tacchella, A.: Quantified Boolean Formulas satisfiability library (QBFLIB) (2005) [www.qbflib.org](http://www.qbflib.org).

## A Proofs

### A.1 Program Encodings

We encode finite-state programs as loop-free imperative programs consisting of a sequence of instructions, each instruction consisting of an opcode and a tuple of operands. The opcode specifies which operation is to be performed and the operands are the arguments on which the operation will be performed. We allow an operand to be one of: a constant literal, an input to the program, or the result of some previous instruction. Such a program has a natural correspondence with a combinational circuit.

A sequence of instructions is certainly a natural encoding of a program, but we might wonder if it is the *best* encoding. We can show that for a reasonable set of instruction types (i.e. valid opcodes), this encoding is optimal in a sense we will now discuss. An encoding scheme  $E$  takes a function  $f$  and assigns it a name  $s$ . For a given ensemble of functions  $F$  we are interested in the worst-case behaviour of the encoding  $E$ , that is we are interested in the quantity

$$|E(F)| = \max\{|E(f)| \mid f \in F\}$$

If for every encoding  $E'$ , we have that

$$|E(F)| = |E'(F)|$$

then we say that  $E$  is an *optimal encoding* for  $F$ . Similarly if for every encoding  $E'$ , we have

$$O(|E(F)|) \subseteq O(|E'(F)|)$$

we say that  $E$  is an *asymptotically optimal encoding* for  $F$ .

**Lemma 1 (Languages with ITE are Universal and Optimal Encodings for Finite Functions).** *For an imperative programming language including instructions for testing equality of two values (EQ) and an if-then-else (ITE) instruction, any total function  $f : S \rightarrow S$  can be computed by a program of size  $O(|S| \log |S|)$  bits.*

*Proof.* The function  $f$  is computed by the following program:

```

t1 = EQ(x, 1)
t2 = ITE(t1, f(1), f(0))
t3 = EQ(x, 2)
t4 = ITE(t3, f(2), t2)
...
```

Each operand can be encoded in  $\log_2(|\mathcal{S}| + l) = \log_2(3 \times |\mathcal{S}|)$  bits. So each instruction can be encoded in  $O(\log |\mathcal{S}|)$  bits and there are  $O(|\mathcal{S}|)$  instructions in the program, so the whole program can be encoded in  $O(|\mathcal{S}| \log |\mathcal{S}|)$  bits.

**Lemma 2.** *Any representation that is capable of encoding an arbitrary total function  $f : \mathcal{S} \rightarrow \mathcal{S}$  must require at least  $O(|\mathcal{S}| \log |\mathcal{S}|)$  bits to encode some functions.*

*Proof.* There are  $|\mathcal{S}|^{|\mathcal{S}|}$  total functions  $f : \mathcal{S} \rightarrow \mathcal{S}$ . Therefore by the pigeonhole principle, any encoding that can encode an arbitrary function must use at least  $\log_2(|\mathcal{S}|^{|\mathcal{S}|}) = O(|\mathcal{S}| \log |\mathcal{S}|)$  bits to encode some function.

From Lemma 1 and Lemma 2, we can conclude that *any* set of instruction types that include ITE is an asymptotically optimal function encoding for total functions with finite domains.

## A.2 Complexity of Finite State Program Synthesis

**Theorem 9 (Second-Order SAT is Polynomial Time Reducible to Finite Synthesis).** *Every instance of Definition 3 is polynomial time reducible to an instance of Definition 4.*

*Proof.* We first Skolemise the instance of definition 3 to produce an equisatisfiable second-order sentence with the first-order part only having universal quantifiers (i.e. bring the formula into Skolem normal form). This process will have introduced a function symbol for each first order existentially quantified variable and taken linear time. Now we just existentially quantify over the Skolem functions, which again takes linear time and space. The resulting formula is an instance of Definition 4.

## A.3 Soundness and Completeness

**Theorem 10.** *Algorithm 1 is sound – if it terminates with witness  $P$ , then  $P \models \sigma$ .*

*Proof.* The procedure SYNTH terminates only if SYNTH returns “valid”. In that case,  $\exists x. \neg \sigma(P, x)$  is unsatisfiable and so  $\forall x. \sigma(P, x)$  holds.

**Theorem 11.** *Algorithm 1 is semi-complete – if a solution  $P \models \sigma$  exists then Algorithm 1 will find it.*

*Proof.* If the domain  $X$  is finite then the loop in procedure SYNTH can only iterate  $|X|$  times, since by this tProofsime all of the elements of  $X$  would have been added to the inputs set. Therefore if the SYNTH procedure always terminates, Algorithm 1 does as well.

Since the EXPLICITSEARCH routine enumerates all programs (as can be seen by induction on the program length  $l$ ), it will eventually enumerate a program that meets the specification on whatever set of inputs are currently being tracked,

since by assumption such a program exists. Since the first-order theory is decidable, the query in VERIF will succeed for this program, causing the algorithm to terminate. The set of correct programs is therefore recursively enumerable and Algorithm 1 enumerates this set, so it is semi-complete.

**Theorem 12.** *Algorithm 1 with the stopping condition described in Section 5.7 is complete when instantiated with  $C^-$  as a background theory – it will terminate for all specifications  $\sigma$ .*

*Proof.* If the specification is satisfiable then Theorem 11 holds, and if it is not then the stopping condition will eventually hold at which point we (correctly) terminate with an “unsatisfiable” verdict.